miASMa: A 2 pass Macro Assembler for x86 (Assembler Design)

Hareesh Nagarajan

Pramod Kumar T

RV03CS6P19

Department of Computer Science & Engineering

R.V. College of Engineering, Bangalore

12 June, 2003

Contents

1 INTRODUCTION **1**

1.1 WHAT IS 2 PASS? 2

1.2 WHAT IS A MACRO PROCESSOR? 3

1.3 WHAT ARE RELOCATABLE FILES WHICH CONFORM TO THE ELF FORMAT? 4

2 COMPARISON BETWEEN NASM AND miASMa **4**

3 SOFTWARE REQUIREMENTS SPECIFICATION **5**

3.1 INTRODUCTION 5

3.1.1 PURPOSE 5

3.1.2 SCOPE 6

3.1.3 DEFINITIONS, ACRONYMS AND ABBREVIATIONS 6

3.1.4 REFERENCES 6
3.1.5 OVERVIEW 6
3.2 GENERAL DESCRIPTION 6
3.2.1 PRODUCT PERSPECTIVE 6
3.2.2 PRODUCT FUNCTIONS 7
3.2.3 USER CHARACTERISTICS 7
3.2.4 GENERAL CONSTRAINTS 7
3.2.5 ASSUMPTIONS AND DEPENDENCIES 7
3.3 FUNCTIONAL REQUIREMENTS 7
3.3.1 INTRODUCTION 7
3.3.2 INPUT SPECIFICATIONS 8
3.3.3 OUTPUT SPECIFICATIONS 9
3.3.4 PROCESSING SPECIFICATIONS 9
3.4 EXTERNAL INTERFACE REQUIREMENTS 9
3.4.1 USER INTERFACE REQUIREMENTS 9
3.4.2 HARDWARE INTERFACE REQUIREMENTS 10
3.4.3 SOFTWARE REQUIREMENTS 10
3.5 PERFORMANCE SPECIFICATIONS 10
3.6 DESIGN CONSTRAINTS 10
3.6.1 STANDARDS COMPLIANCE 10
3.6.2 HARDWARE LIMITATIONS 10

4 SYSTEM DESIGN - Design Document 11
4.1 ARCHITECTURAL DESIGN 11
4.1.1 PROBLEM SPECIFICATION 11
4.1.2 ENTITY RELATIONSHIP DIAGRAM 11
4.1.3 DATA FLOW DIAGRAMS 11
4.1.4 STRUCTURE CHART 12
4.1.5 MODULE SPECIFICATIONS 12
4.1.6 MODULE-0 14

4.1.7 MODULE-1 . . . 14
4.1.8 MODULE-2 . . . 14
4.1.9 MODULE-3 . . . 15
4.1.10 MODULE-4 . . . 15
4.1.11 MODULE-5 . . . 15
4.1.12 MODULE-6 . . . 16
4.1.13 MODULE-7 . . . 16
4.1.14 MODULE-8 . . . 16
4.1.15 MODULE-9 . . . 16
4.1.16 MODULE-10 . . . 17
4.1.17 MODULE-11 . . . 17
4.1.18 MODULE-12 . . . 17
4.1.19 MODULE-13 . . . 18
4.2 DETAILED DESIGN . . . 18
4.2.1 DESIGN DECISIONS . . . 18
4.2.2 DATA DEFINITIONS/DICTIONARY . . . 18

5 A Demonstration of Scanning and Parsing in miASMa 18

6 TOOLS USED TO BUILD MIASMA 20
6.1 LEX . . . 21
6.1.1 FUNCTION IN MIASMA . . . 21
6.2 YACC . . . 22
6.2.1 FUNCTION IN MIASMA . . . 22
6.3 GNU ld . . . 24
6.3.1 FUNCTION IN MIASMA . . . 24
6.4 Qt . . . 25
6.4.1 FUNCTION IN MIASMA . . . 25

7 UNDERSTANDING THE ELF FORMAT 25
7.1 WITH READELF . . . 26

7.2 WITH OBJDUMP 29
7.3 WITH A HELLO.O MAP 31
7.4 A LOOK AT THE FINAL EXECUTABLE 33

8 libmiASMaelf LIBRARY 35

9 IMPLEMENTATION 39
9.1 EXAMPLE PROGRAMS THAT CAN BE ASSEMBLED WITH MIASMA 39
9.1.1 BUBBLE SORT 39
9.1.2 FACTORIAL 40
9.1.3 FIBONACCI SERIES 42
9.1.4 nCr COMBINATION 43
9.2 SCREEN SHOTS 45

10 SYSTEM TESTING 45
10.1 UNIT TEST REPORT 47
10.1.1 DRIVERS AND STUBS 47
10.2 TEST SUMMARY 47
10.2.1 MODULE-0 47
10.2.2 MODULE-1 47
10.2.3 MODULE-2 48
10.3 SYSTEM TEST REPORTS 48
10.4 ERROR REPORTS 48

11 RESULTS AND CONCLUSION 49

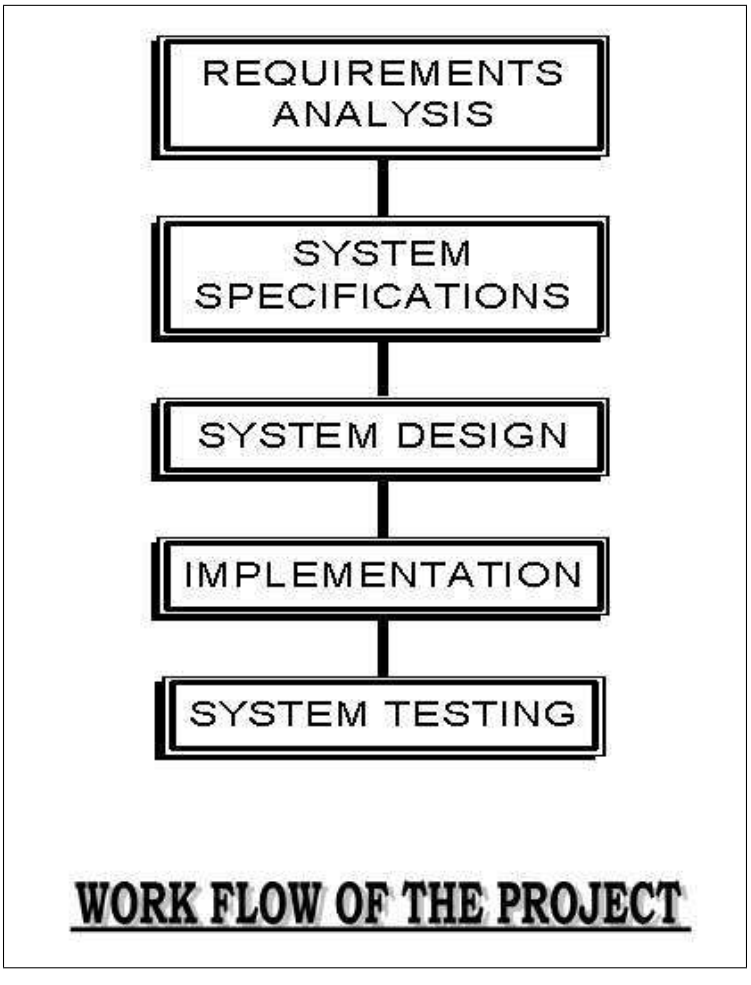

Figure 1: Work Flow

1 INTRODUCTION

Before we start with a formal introduction to **miASMa**, let us view 1 the outline of the development processes which we were involved with, during the development of **miASMa**.

miASMa is an original, independently conceived, easy to use, x86 assembler written for the Linux platform. On the Linux platform barring NASM and more recently YASM there aren't many easy to use x86 assemblers. Our aim was to write an assembler by which we would understand the various machine dependent features[1] as well as operating

[1]Intel Instruction Set

specific features[2] which is the crux of designing an assembler.
Let us now dissect the following description of **miASMa**:
miASMa is A 2 pass Macro Assembler for an x86 machine which generates Relocatable files that conform to the ELF Format.

1.1 WHAT IS 2 PASS?

A 2 Pass assembler is one that goes through the source file (Eg: hello.asm) two times. It does so, so that the addresses of any forward declaration which it encounters during the first pass is resolved during the second pass. If the declaration is not resolved an error is flagged. E.g: Let us look at an assembly program snippet

```
SEGMENT .DATA
VAR DB 'HELLO'
...
SEGMENT .TEXT
MOV AB,VAR
J LABEL
...
...
```

VAR has been defined as shown, hence VAR has an address associated with it which is known during the first pass. In the first pass the MOV instruction can be assembled without a hitch, but the instruction J LABEL cannot be assembled because the variable LABEL has not been defined at that point of time. Nonetheless, the definition of LABEL **may** be found later in the first pass, at which point it is too late for the instruction J LABEL to be assembled. Therefore, **miASMa** makes a secondary pass over the source file and resolves the addresses of every label in the program.
If the variable has not been defined an error is flagged.

[2]Object Formats. In our case it is the ELF Format

1.2 WHAT IS A MACRO PROCESSOR?

If we wanted to write a program which printed a number of messages on the screen, then one would have to repeatedly execute these steps over and over again:

```
...
MOV EAX,4
MOV EBX,1
MOV ECX,STRING1
MOV EDX,LENGTH_OF_THE_STRING1
INT 80H
...
...
MOV EAX,4
MOV EBX,1
MOV ECX,STRING2
MOV EDX,LENGTH_OF_THE_STRING2
INT 80H
...
...
```

Now with the help of a macro processor one just has to create a macro and invoke the macro as and when we want to print a message.

```
/**** The Complete Program ***/
MACRO SCALL A B C D
MOV AX,A
MOV BX,B
MOV ECX,C
MOV DX,D
INT 80H
ENDM
```

```
SEGMENT .DATA
ABC DB 'HELLOWORLD'
DEF DB 'MIASMAROCKS'
SEGMENT .TEXT
SCALL 4 1 ABC 10
SCALL 4 1 DEF 11
MOV AX,1
INT 128
```

1.3 WHAT ARE RELOCATABLE FILES WHICH CONFORM TO THE ELF FORMAT?

As mentioned in the abstract, relocatable files hold code and data suitable for linking with other object files so as to create either an executable or a shared object file. Now, one must adhere to the ELF format[3] if we want the Linux operating system to load and execute our object files. One cannot just transliterate an instruction such as
MOV EAX,4
to a text file which the following characters **B8 04 00 00 00**, and expect the OS to load and execute the text file. In such a case the behavior of **ld** and the **operating system** is undefined.
Hence there exists a certain format which one must conform to, to write the object file. This format is as specified by TIS-Tool Interface Standard[1]. We later discuss **libmiasmaelf** a library which is used by **miASMa** to build the relocatable files.

2 COMPARISON BETWEEN NASM AND miASMa

Figure 2 presents a comparision of assembly times between NASM - Net Wide Assembler and **miASMa**. NASM beats **miASMa** in speed, primarily because it uses a custom built

[3]One can of course recompile the kernel to support other object files

Time Comparison between NASM and miASMa

ASSEMBLER	FILE	MACRO	PASS1	PASS2	CUMULATIVE TIME
miASMa	Hello.asm	0.005 s	0.006 s	0.006 s	0.017 s
NASM	Hello.asm	-	-	-	0.015 s
miASMa	Fibo.asm	0.005 s	0.006 s	0.006 s	0.017 s
NASM	Fibo.asm	-	-	-	0.020 s

NOTE: 1.Fibo.asm computes Fibonacci series up to 2 digit numbers (mod 100)

2. Since miASMa divides assembly into 3 parts, we have shown them separately instead of invoking the entire procedure through a shell script, which would create additional overhead.
3. All times are measured in seconds. They are the Real Times
4. The above comparison was performed on a Pentium III – 650 MHz running Red Hat 7.3.
5. All tests were performed at the same system load.

Figure 2: Comparison between NASM and miASMa

Lexer and Parser. A comprehensive set of tests have not been performed as yet, it will be done in the near future.

3 SOFTWARE REQUIREMENTS SPECIFICATION

Given below is a formal Software Requirements Specification based on a template as prescribed in Jalote[3]

3.1 INTRODUCTION

3.1.1 PURPOSE

The main purpose of this project is to design and implement an assembler for an INTEL 8086 machine. It is supposed to perform fundamental functions like translating mnemonic

operation codes to their machine language equivalents and assigning machine addresses to symbolic labels and creation of a object file which can be linked.

3.1.2 SCOPE

This is basically an educational project and hence will help in understanding the working of assemblers and various concepts like object code formats (ELF,COFF etc),object code generation and the loading and linking processes.

3.1.3 DEFINITIONS, ACRONYMS AND ABBREVIATIONS

NASM - Net wide Assembler. ELF - Executable and Linkable Format. COFF - Common Object File Format.

3.1.4 REFERENCES

Information regarding the references made in the document are presented in the Bibliography at the end.

3.1.5 OVERVIEW

In LINUX, most of the assemblers are closely integrated with compilers. NASM is the only independent assembler. This product will be designed with a constant goal of maintaining software visibility and simplicity.

3.2 GENERAL DESCRIPTION

3.2.1 PRODUCT PERSPECTIVE

This product will be designed to produce a linkable object file that can sent as input to the GNU linker LD.

3.2.2 PRODUCT FUNCTIONS

The assembler scans the input program and groups the characters into pre-defined tokens. These tokens are later parsed to match the assembly language constructs. Depending on the match recognized by the parser, the appropriate object code is written into an object file. A macro processor module that runs prior to the assembler expands all user defined macros.

3.2.3 USER CHARACTERISTICS

The user is an assemble language programmer who is provided with an editor-like interface which allows him to edit and assemble programs. The user can also view the object code generated and intermediate files like the output of the microprocessor.

3.2.4 GENERAL CONSTRAINTS

Only a subset of INTEL 8086 instruction set have been implemented. There is scope for expansion of the instruction set and the addressing modes supported by the assembler.

3.2.5 ASSUMPTIONS AND DEPENDENCIES

It has been assumed that the operating system has a linker that can link ELF object files.

3.3 FUNCTIONAL REQUIREMENTS

3.3.1 INTRODUCTION

The system specification process lies between the steps of requirement analysis and system design. The requirement document is primarily concerned with the end users view of the system and is written at a high level with the less important details omitted. These details must be supplied to provide a basis for system design. These specifications must contain the complete description of all the functions and constraints of the desired software system. They must be clearly and precisely written, must be consistent and must contain all the information needed to write and test the software. In order to create such specifications,

the developers must examine the purpose and the goals of the system more closely. The process of formulating precise system specification often reveals areas where requirements are incomplete or ambiguous.

3.3.2 INPUT SPECIFICATIONS

1 The labels in the source program, if present must begin on a new line. The opcode field should be separated from the label field by a colon. If no label is present, the opcode may begin on a newline.

2 Labels should satisfy the following requirements. The first character must be an alphabet or underscore (_a-zA-Z) and each of the remaining characters can be alphabets or Numbers. Label names should not match with keywords or reserved words (opcodes and assembler directives).

3 The opcode field must be from the instruction set supported or from one of the assembler directives (Pseudo instructions).

4 An instruction operand may be either a symbol which appears as a label in the program or a number that represents an immediate operand or an INTEL 8086 register.

5 The source program should contain distinct labels.

6 All macros should be defined prior to the beginning of the any segment.

To Summarize

- Specification 1 deals with the format of the input, describing how the subfields of a source statement are positioned.

- Specification 2 gives the rules for the formation of labels.

- Specification 3 describes the set of entries that are allowed to occur in a particular subfield, thus giving constraints on the contents of the input.

- Specification 4 provides the context dependent constraints.

- Specification 5 provides implementation constraints

- Specification 6 provides constraints on the position of the macros.

3.3.3 OUTPUT SPECIFICATIONS

1 The assembly listing should show each source program statement, together with the current location value, object code generated or any error messages . This describes the form and contents of the desired output.

2 The object program should occupy no address out of a particular range or limit. This specifies the constraints under which output values are determined.

3 The object program should not be generated if any assembly errors are reported. This specifies the conditions under which the output is generated.

4 The most important constraint is that it should be in the ELF format.

3.3.4 PROCESSING SPECIFICATIONS

1 The scanner should scan the input program and group the characters into predefined tokens. This process of scanning should not only identify tokens but also classify them as one of the following : opcode,register,symbol or immediate operand.

2 The parser should have the complete specification of the assembly language constructs (grammar), so that it can match instructions correctly.

3.4 EXTERNAL INTERFACE REQUIREMENTS

3.4.1 USER INTERFACE REQUIREMENTS

The user interface is keyboard and mouse based.

3.4.2 HARDWARE INTERFACE REQUIREMENTS

INTEL 80X86 or above processors.

3.4.3 SOFTWARE REQUIREMENTS

PLATFORM: UNIX/LINUX **SOFTWARE TOOLS:**

- Scanner(Lexer) Generator like LEX
- Parser Generator like YACC
- GNUs Binary Tools that comprises of tools such as Obj dump, readelf etc
- C++ compiler like G++
- Linking Loader like LD

3.5 PERFORMANCE SPECIFICATIONS

1 The assembler should be able to process at least 25 lines of source statement per second of computer time.

2 The user should be able to understand the error messages easily.

3 The assembler should fail to process the source programs correctly in no more than 0.01% percent of all executions.

3.6 DESIGN CONSTRAINTS

3.6.1 STANDARDS COMPLIANCE

miASMa is does not adhere to every specification specified in the TIS Document.

3.6.2 HARDWARE LIMITATIONS

None.

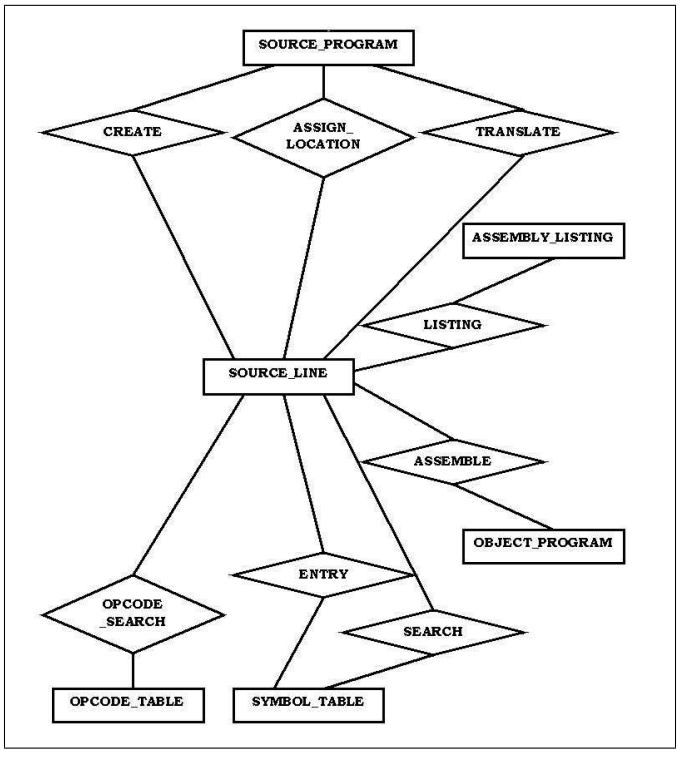

Figure 3: ER Diagram for **miASMa**

4 SYSTEM DESIGN - Design Document

4.1 ARCHITECTURAL DESIGN

4.1.1 PROBLEM SPECIFICATION

The problem on hand is to write a 2 pass macro assembler for any Intel x86 machine that generates object files that comply with the ELF format.

4.1.2 ENTITY RELATIONSHIP DIAGRAM

Refer Figure 3

4.1.3 DATA FLOW DIAGRAMS

Refer Figure 4 for 0 Level DFD

Refer Figure 5 for 1 Level DFD

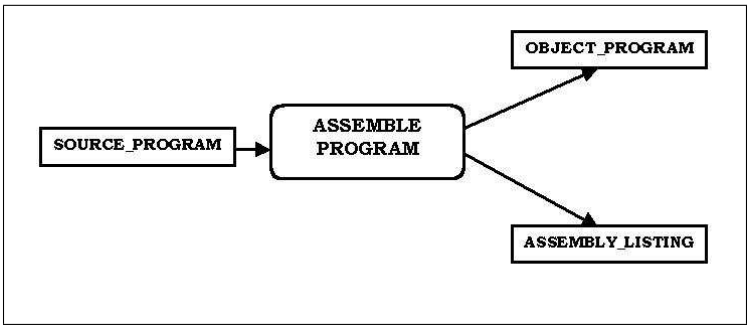

Figure 4: 0 Level DFD

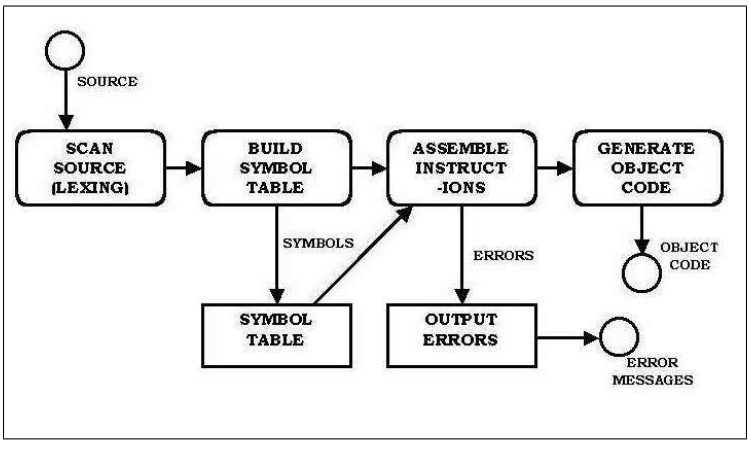

Figure 5: 1 Level DFD

Refer Figure 6 for 2 Level DFD

4.1.4 STRUCTURE CHART

Refer Figure 7

4.1.5 MODULE SPECIFICATIONS

A number of the modules mentioned below are generic to most assemblers, but have been written specifically for **miASMa**. Refer beck[2].

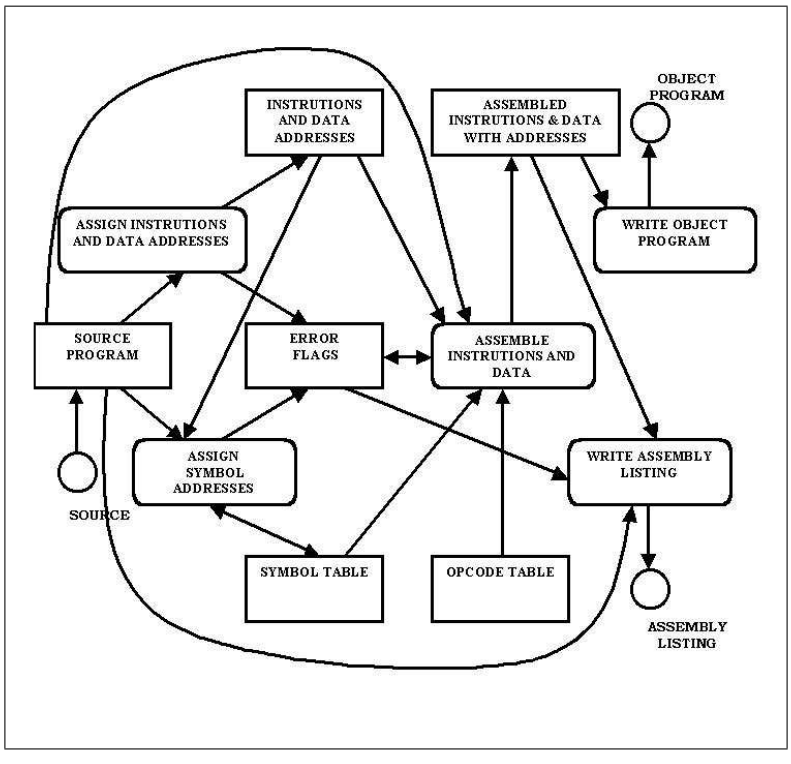

Figure 6: 2 level DFD

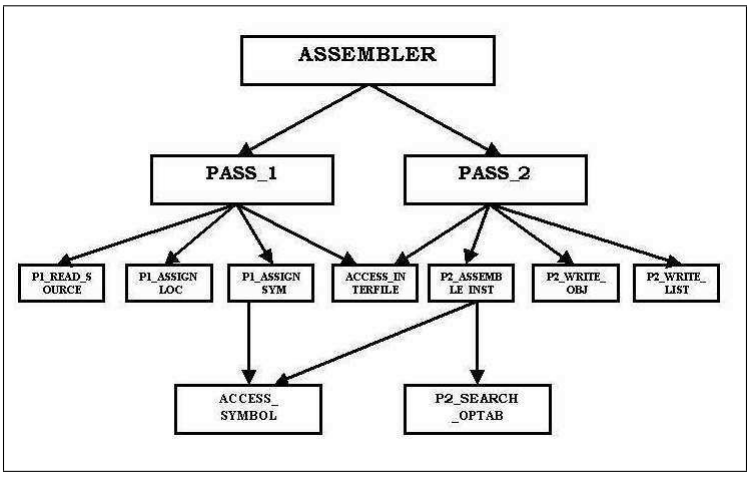

Figure 7: Structure Chart

4.1.6 MODULE-0

NAME: miasma

INPUT: Source program file to be assembled.

OUTPUT: In case of sauces, the output of the program is displayed. In addition, files like output of the macro preprocessor, the symbol file and the object code generated. In case of failure, the reported errors are displayed with their complete specification.

CALLED BY: User

SUBORDINATES: macro preprocessor, pass_1 and pass_2 (calls each of these only once).

PURPOSE: To assemble and execute the source program

4.1.7 MODULE-1

NAME: macro preprocessor

INPUT: Source program file to be assembled.

OUTPUT: File with all the user defined macros expanded in case of success. Otherwise it signals error to the caller.

CALLED BY: miasma.

SUBORDINATES: A lexer called macro which handles the actual expansion of macros.

PURPOSE: To expand the user defined macro into a separate file.

4.1.8 MODULE-2

NAME: pass_1

INPUT: Output file of the macro preprocessor (with all the macros expanded, if any).

OUTPUT: An intermediate symbol file which contains the indices and the location counter values of all the user defined symbols in the source program.

CALLED BY: miasma

SUBORDINATES: A lexer called scan which handles the recognition and classification of the tokens and a parser called pass1_parse which handles the recognition of the scanned tokens as some language construct defined and accordingly updates the symbol file.

PURPOSE: To assign addresses to symbolic labels and record it for the use in pass_2.

4.1.9 MODULE-3

NAME: pass_2
INPUT: Output file of the macro preprocessor and the intermediate symbol file which is the output of the pass_1.
OUTPUT: A file containing the actual object code generated which forms the input to the loader LD.
CALLED BY: miasma
SUBORDINATES: A lexer called scan which handles the recognition and classification of the tokens and a parser called pass2_parse which handles the recognition of the scanned tokens as some language construct defined, generating the object code and accordingly updating the object code file.
PURPOSE: To translate mnemonic operation codes to their machine language equivalents

4.1.10 MODULE-4

NAME: macro
INPUT: Source statement.
CALLED BY: macro preprocessor
PURPOSE: To help the macro preprocessor in scanning the tokens and to write the expanded program to the output file.

4.1.11 MODULE-5

NAME: scan
INPUT: Output file of the macro preprocessor.
OUTPUT: Return code, the source statements and the error flags (if any).
CALLED BY: pass1_parse as well as pass2_parse.

PURPOSE: To group characters in the source file into tokens and to associate appropriate values with the tokens.

4.1.12 MODULE-6

NAME: pass1_parse
INPUT: Output file of the macro preprocessor (with all the macros expanded, if any).
CALLED BY: pass_1
SUBORDINATES: pass1_assign_sym, write_intsym_file
PURPOSE: To recognize the scanned tokens as language constructs and to generate the intermediate symbol file.

4.1.13 MODULE-7

NAME: pass2_parse
INPUT: Output file of the macro preprocessor and the intermediate symbol file which is the output of the pass_1.
CALLED BY: pass_2
SUBORDINATES: read_intsym_file, pass2_assemble_inst,pass2_write_obj.
PURPOSE: To recognize the scanned tokens as language constructs and to generate the object file.

4.1.14 MODULE-8

NAME: write_intsym_file
INPUT: Symbols, indices and the address.
CALLED BY: pass1_parse.
PURPOSE: To store user defined symbols with their addresses for use in pass_2.

4.1.15 MODULE-9

NAME: read_intsym_file
INPUT: Symbols

OUTPUT: Return code and address.

CALLED BY: pass2_parse.

PURPOSE: To read the address of the user defined symbols to be used in generating the object code.

4.1.16 MODULE-10

NAME: pass1_assign_sym.

INPUT: Source statement, symbol and the current LC.

OUTPUT: Error flags (if any).

CALLED BY: pass1_parse.

SUBORDINATES: write_intsym_file.

PURPOSE: To keep track of the location counter and assign addresses to symbols.

4.1.17 MODULE-11

NAME: pass2_assemble_inst.

INPUT: Source statement, current LC.

OUTPUT: Object code or the error flags.

CALLED BY: pass2_parse.

SUBORDINATES: pass2_search_optab.

PURPOSE: To generate the object code for each source line.

4.1.18 MODULE-12

NAME: pass2_search_optab.

INPUT: Mnemonic code.

OUTPUT: Return code and the machine code of mnemonic.

CALLED BY: pass2_assemble_inst.

PURPOSE: To provide the machine code for the mnemonic for each source line.

4.1.19 MODULE-13

NAME: pass2_write_obj.
INPUT: Current LC and the actual object code.
CALLED BY: pass2_parse.
PURPOSE: To write the generated object code into the object file .

4.2 DETAILED DESIGN

4.2.1 DESIGN DECISIONS

The design decisions are mentioned in great details in sections presented later.

4.2.2 DATA DEFINITIONS/DICTIONARY

The data dictionary contains the base object codes for all instructions in the supported instruction set.

5 A Demonstration of Scanning and Parsing in miASMa

Let us consider a sample 8086 instruction. Figure 10 summarizes the entire process that is applicable to **miASMa**. **ADD CX, 28H**
This instruction is first scanned by the lexer i.e. **yylex()** generated by lex. The lexer converts this instruction into a stream of tokens . The token stream for the above instruction will contain the following

1 An ADD token.

2 A space.

3 A REG16 token associated with a value of 1 in yylval to indicate that the register is CX.

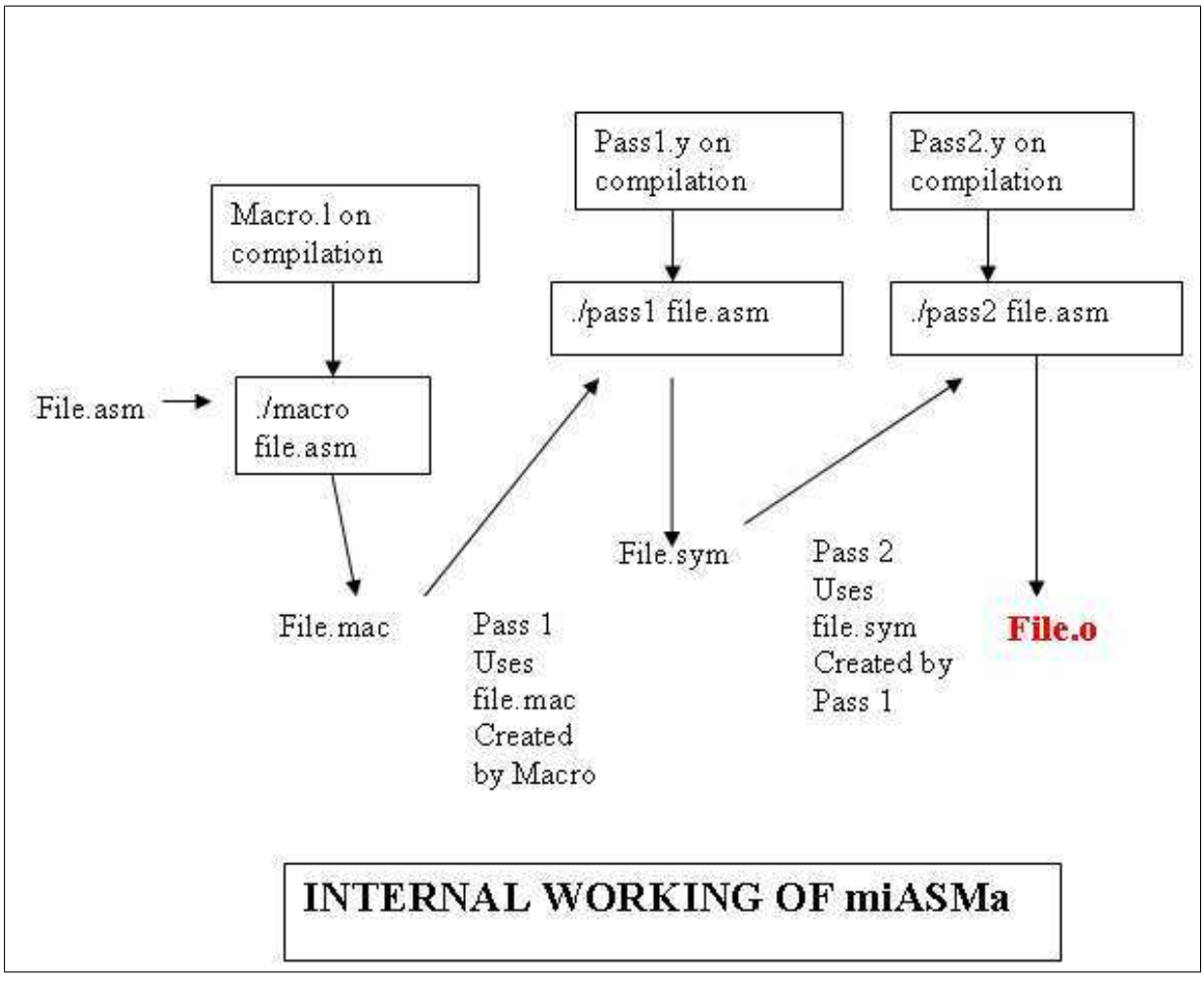

Figure 8: How **miASMa** Works

4 A "," token.

5 A space.

6 An IMM token for "28H" that identifies a decimal or hexadecimal number. This token is associated with a value of 40 (decimal value of 28H) in yylval . This token stream is then passed on to the pass1 parser . The pass1 parser then recognizes this stream as the instruction
ADD REG16, IMM

The action associated with this rule is to increment the location counter of the text section by 5, which is the size (in bytes) of the object code for the above instruction. The value of this location counter is used to assign addresses to the labels used in the text section. When this instruction is encountered in the pass2 parser, it is recognized as the same instruction but different actions are taken.
Firstly, the base object code for the instruction is decided. This contains information about the instruction itself and the addressing mode. Then information about the operands is added to the object code at appropriate positions.

In this instruction the value associated with REG16 is in $2. This value is added to the third byte of the object code to provide information about the register to which IMM is to be added.
The value associated with IMM is in $4 and is the numeric value of the operand. The value is encoded into the last two bytes of the object code. With this the parsing of the instruction is completed. The parser passes on the object code to the code generator which writes the code out to the object file in the required format (Object format in our case is ELF).

6 TOOLS USED TO BUILD MIASMA

In this section we give an introduction to the software tools that were used to create **miASMa**.

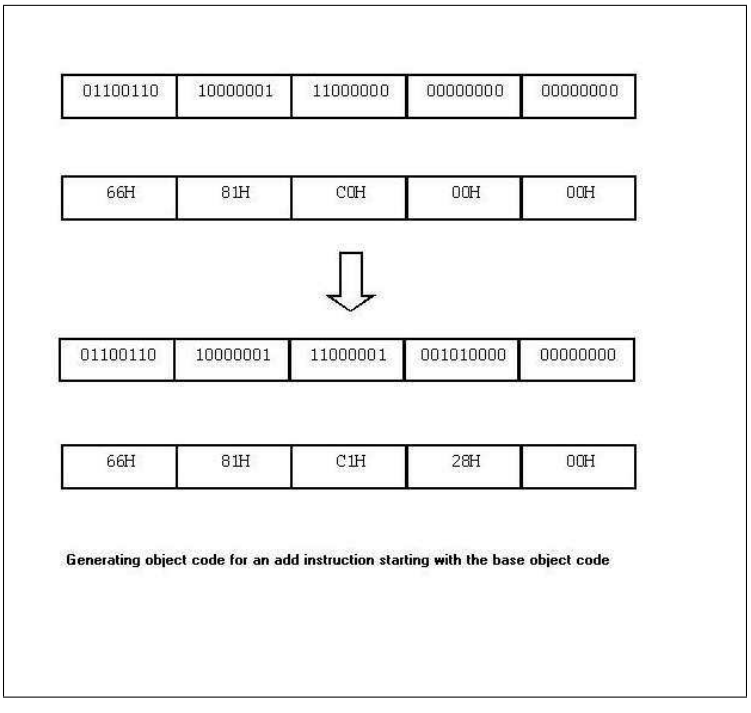

Figure 9: Demonstration of Parsing

6.1 LEX

Lex is officially known as a **Lexical Analyzer**. It's main job is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file(source file) [6].

6.1.1 FUNCTION IN MIASMA

For example, in **miASMa** LEX is used as a syntax analyzer that returns tokens that it finds in the source file. An example of a TOKEN would be the keyword - 'MOV'. As such an 'O' in the source file would be treated as a character, and is identified as an ALPHA by the pattern:

```
DIGIT [0-9]
ALPHA [A-Z]
ALPDI [A-Z0-9]
```

Hence when the Lex sees a "MOV AX,BX" in the source file, it returns the token[4] MOV to the parser, then returns an AX and then a BX. The Parser in turn analyzes what is to be done with the instruction.

The **Lexical rules** for the above instruction are

```
"AX"|"BX"|"CX"|"DX"|"SP"|"BP"|"SI"|"DI"
  {
    if(!strcmp(yytext,"AX"))yylval = 0;
    ...
    "MOV "
    {
      return MOV;
    }
    ...
```

6.2 YACC

Yacc is officially known as a **parser**. It's job is to Analise the structure of the input stream, and operate on the "big picture". In the course of it's normal work, the parser also verifies that the input is syntactically sound. It is possible to create a simple parser using Lex alone. by making extensive use of the user-defined states (ie start-conditions). However, such a parser quickly becomes unmaintainable, as the number of user-defined states tends to explode[6].

6.2.1 FUNCTION IN MIASMA

First Pass

The YACC Routines for the instruction MOV AX,BX in the first Pass of **miASMa** are:

```
mov:...
```

[4]LEX tries to find the largest token

```
|MOV REG16 "," REG16     {
TextLocCtr += 2;
 }
```

TextLocCtr is the Text Location Counter for instructions in the text section. As a **MOV REG16, REG16** instruction is 2 bytes long [5] we increment the Text Location Counter by 2 bytes.

So as we can see the YACC routines above makes sense of the tokens and perform the action of incrementing the Text Location Counter.

Second Pass In the second pass of **miASMa** the same grammar is interpreted again but the action associated with it is different. At this point machine code is generated. The function func acts as a wrapper function into the **libmiasmaelf** library which generates a relocatable file that conforms to the ELF Format. The library will be explained later in the document.

```
mov: ...
|MOV REG16 "," REG16
{
char a[2] = {'\x89','\xC0'};
a[1] +=($4<<3);
a[1] +=$2;
func(a,sizeof(a));
TextLocCtr += 2;
}
....
....
%%
void func(char* text,unsigned int size )
{
```

[5] AX, BX are 16 Bit Registers

```
if(TextDefined)
{
vector<char> vtext;
for(int i = 0;i < size;++i)
vtext.push_back(text[i]);
obj.AddContents(vtext,
obj.GetSectionIndexOfType(SHT_PROGBITS,".text"));
}
else
printf("Miasma: .text Not Defined ");
}
```

6.3 GNU ld

ld combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run ld. ld reads, combines, and writes object files in many different formats–for example - COFF, a.out, ELF etc. Different formats may be linked together to produce any available kind of object file[6]. Aside from its flexibility, the GNU linker is more helpful than other linkers in providing diagnostic information. Many linkers abandon execution immediately upon encountering an error; whenever possible, ld continues executing, allowing you to identify other errors (or, in some cases, to get an output file in spite of the error)[6].

6.3.1 FUNCTION IN MIASMA

Since **miASMa** primarily(version 0.1) generates only Relocatable files that conform to the ELF format, and ld needs to be used ONLY to create the executable file from the relocatable file.

For example:

[6]This is the case as we shall see later on, when we forget to add the _start symbol to our object file

```
\$miasma hello.asm -o hello.o
\$ld hello.o -o hello
\$./hello
Hello World!
\$
```

As of now **miASMa** does not support, the linking of multiple files i.e external symbol references, hence ld's invocation is restricted to the above command. It must be mentioned here, that **miASMa** barely capitalizes on the power of GNU's ld.

6.4 Qt

Qt 3 is a multi platform C++ application development framework. Its primary strength is its Short Learning curve since Qt Programmers have to learn a single API to write apps that run almost anywhere. Qt also has a rich set of standard widgets, and lets one write custom controls[5].

6.4.1 FUNCTION IN MIASMA

Qt is used for building the IDE. The IDE displays

- Any errors that might have occurred during the Assembly of the source file
- The object dump of the relocatable file created
- The ELF information of the relocatable file generated

We highly recommend the usage of a terminal based editor such as vi or emacs to write assembly programs, primarily because they are not as easy to use as vi or emacs!

7 UNDERSTANDING THE ELF FORMAT

Before we get into the actual ELF format, we made a crucial point in the previous section which was - the relocatable file that **miASMa** generates is converted into the executable

file, by invoking ld.

7.1 WITH READELF

Let us take an example of a simple assembly program: hello.asm to help us in understanding the format.

```
SEGMENT .DATA
myvariable DB 'Hello, World!'
SEGMENT .TEXT
MOV EAX,4
MOV EBX,1
MOV ECX,myvariable
INT 80H
MOV EAX,1
INT 80H
```

On assembling hello.asm we obtain **hello.o** which is the relocatable file **miASMa** creates. Let us see the ELF Structure of this file using **readelf** a tool that is bundled with GNU Binary Utilities.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
```

```
  Start of program headers:          0 (bytes into file)
  Start of section headers:          52 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         7
  Section header string table index: 2

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .strtab           STRTAB          00000000 00014c 000013 00      0   0  1
  [ 2] .shstrtab         STRTAB          00000000 00015f 000031 00      0   0  0
  [ 3] .text             PROGBITS        00000000 000190 00001d 00  AX  0   0 16
  [ 4] .data             PROGBITS        00000000 0001b0 00000e 00  WA  0   0 16
  [ 5] .symtab           SYMTAB          00000000 0001c0 000030 10      1   1  4
  [ 6] .rel.text         REL             00000000 0001f0 000008 08      5   3  4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no program headers in this file.

Relocation section '.rel.text' at offset 0x1f0 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000b  00000208 R_386_RELATIVE    00000000   myvariable
```

```
There are no unwind sections in this file.

Symbol table '.symtab' contains 3 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    WEAK   DEFAULT    3 _start
     2: 00000000     0 OBJECT  GLOBAL DEFAULT    4 myvariable

No version information found in this file.
```

As we can see the first part of the output constitutes the ELF header. The header contains a Magic number in the beginning which is used to detect, if the object file is a valid ELF file or not. There are other attributes as well, which can be obtained from the documentation[1].
The Number of section headers refers to the number of sections present in the object file. The Section header string table refers to the header of **.shstrtab** section. The contents of this section comprise of the names of the various other sections in the file. On careful observation of the earlier output, we infer that the contents of this section will be:

```
{'\0','.','s','t','r','t','a','b','\0','.','s','h','t','r','t',
'a','b','\0'........
and so on upto ,'r','e','l','.','t','e','x','t',\0'}
```

If one did the math, the size of the section is 0x31h.
Similarly we have other sections, such as .text or .data which contain the instructions in machine code (mentioned earlier). By writing a SEGMENT .DATA we create a .DATA section.
Now the headers of each section contain critical attributes regarding the interpretation of the information from the relocatable file. A few of these are

- Offset of the contents from the beginning of the file

- The size of the section

- The alignment of the section

- Other critical flags - For assigning the scope of a variable etc.

It must be mentioned here that **miASMa** uses **libmiASMaelf** for generating the relocatable files, and that **miASMa** does not interpret the scope of any variable. This is primarily because **miASMa** does not support dynamic linking.
Later in the output of readelf above, we observe the contents of the SYMTAB. The contents of this section are explicitly specified in the documentation. We can see that the variable **myvariable** has an **NDX** value of 4. This just means that the variable is defined to the **.data** section, and 4 refers to **.data** section header index. Similarly **_start** is a symbol that miasma generates automatically, because this symbol is needed by ld - as ld assigns _start as the starting address for the program and the rest of the program is addressed relative to it[7].
Somewhere in the middle of readelf's output we see the contents of the relocation section **.rel.text** which exists at an offset of 0x1f0 from the starting of the file. Now at an offset of 11 from the beginning of the file, relocation must be performed. Byte 11 falls in the instruction **MOV ECX, myvariable**, now as the address of **myvariable** needs to be relocated during runtime[8], the section **.rel.text** exists precisely for this purpose.

7.2 WITH OBJDUMP

Another tool provided in the GNU Binary utilities is **objdump**. This program reads the .o file (in this case) and tries to reconstruct the instructions that might have produced the machine code. The alignment of the sections being read is very critical here *For example: If the alignment of a section is set to 2 Bytes and we are interpreting the section as one with an alignment of 4 Bytes then the disassembly, which is essentially what* ***objdump***

[7]Even without the _start symbol ld will create an executable by issuing a warning
[8]Because prior to run time the address is not known

does will be undefined. Thus one must not be too startled when the objdump does not correlate with the expected results.

```
hello.o:     file format elf32-i386


Disassembly of section .text:


00000000 <\_start>:
   0: b8 04 00 00 00        mov    $0x4,%eax
   5: bb 01 00 00 00        mov    $0x1,%ebx
   a: b9 00 00 00 00        mov    $0x0,%ecx
   f: ba 0e 00 00 00        mov    $0xe,%edx
  14: cd 80                 int    $0x80
  16: b8 01 00 00 00        mov    $0x1,%eax
  1b: cd 80                 int    $0x80
Disassembly of section .data:


00000000 <myvariable>:
   0: 48                    dec    %eax
   1: 65                    gs
   2: 6c                    insb   (%dx),%es:(%edi)
   3: 6c                    insb   (%dx),%es:(%edi)
   4: 6f                    outsl  %ds:(%esi),(%dx)
   5: 2c 20                 sub    $0x20,%al
   7: 57                    push   %edi
   8: 6f                    outsl  %ds:(%esi),(%dx)
   9: 72 6c                 jb     77 <_start+0x77>
   b: 64 21 0a              and    %ecx,%fs:(%edx)
```

The series of 11 characters that one sees in the disassembly of .data section is nothing but "Hello,World!". To the right we can see the instructions that objdump has tried to figure

out of "Hello,World!" and they do not make any sense. This is because objdump has no idea if the machine code that it is interpreting is an instruction or if it is an address.

7.3 WITH A HELLO.O MAP

Let us try and correlate the 10 with the actual hello.o file.

- The size of the ELF Header is 52 bytes
- The size of every Section Header is 40 bytes
- The first header is a NULL header which has an offset of 0 and size of 0. This header is mandatory.
- **.strtab** header follows next. This sections contents falls at an offset of 332 bytes from the beginning of the file. The section contents size is at 19 bytes.
- **.shstrtab** similarly has offset=351 and size=49. If we add all the characters of each of the section names with appropriate NULLs filled in ,we obtain a size of 49.
- **.text** section contents have an offset of 400 and a size=29. If we calculate the size of the instructions in SEGMENT .TEXT of hello.asm we will get identical results.
- **.data** falls next at an address of 432 and a size=14. But according to our calculations .data must fall at the address 400+29=429. But this does not happen, if it does then the results are undefined. **(429%16!=0)** A sections **Address Alignment** is specified in the header of that section, and is crucial information. If set incorrectly the OS will fail to execute the program, thus making the results undefined. In this section the alignment is set to 16 bytes.
- **.symtab** similarly follows with an offset of 448 instead of 446. It has a size of 48=(sizeof of Symbol Table Entry structure * Number of Symbols). The sizeof the entry is a fixed 16 bytes, and as we have 3 entries we arrive at a size of 48 bytes.

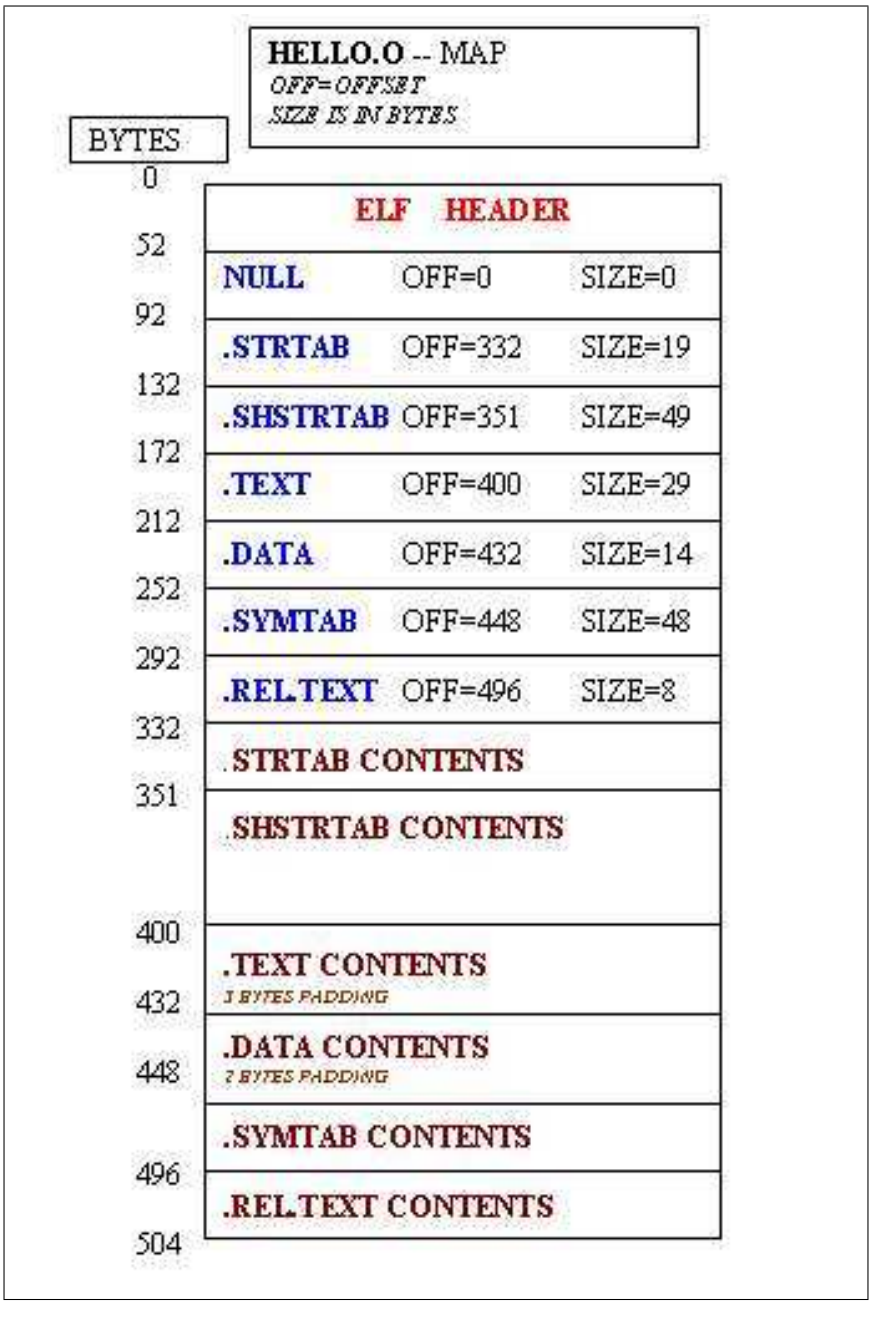

Figure 10: hello.o - Object File Memory Map

- **.rel.text** sections contents fall at an offset of 496 and the size=8 bytes. This section too has a fixed Relocation entry size.

From the above we observe that the size of the file must be 504 bytes and our observation is indeed correct!

```
-rw-r--r--    1 root     root          504 Jun 12 14:18 hello.o
```

7.4 A LOOK AT THE FINAL EXECUTABLE

As we have reiterated over and over again **miASMa** only produces Relocatable files, but Relocatable files are of no use on their own. We invoke **ld** to create the executable. On invoking ld and executing readelf on the file produced, we observe the following ELF Structure. Notice the difference between the **Relocatable File** and **Executable File**. This is all we will have to say about executable files.

```
#readelf hello -a
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048080
  Start of program headers:          52 (bytes into file)
  Start of section headers:          220 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
```

```
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         7
  Section header string table index: 4

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08048080 000080 00001d 00  AX  0   0 16
  [ 2] .data             PROGBITS        080490a0 0000a0 00000e 00  WA  0   0 16
  [ 3] .bss              PROGBITS        080490ae 0000ae 000002 00   W  0   0  1
  [ 4] .shstrtab         STRTAB          00000000 0000b0 00002c 00      0   0  1
  [ 5] .symtab           SYMTAB          00000000 0001f4 0000c0 10      6   7  4
  [ 6] .strtab           STRTAB          00000000 0002b4 000024 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0009d 0x0009d R E 0x1000
  LOAD           0x0000a0 0x080490a0 0x080490a0 0x0000e 0x0000e RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
```

```
There is no dynamic segment in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048080     0 SECTION LOCAL  DEFAULT    1
     2: 080490a0     0 SECTION LOCAL  DEFAULT    2
     3: 080490ae     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    6
     7: 080490a0     0 OBJECT  GLOBAL DEFAULT    2 msg
     8: 08048080     0 FUNC    WEAK   DEFAULT    1 _start
     9: 080490ae     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    10: 080490ae     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
    11: 080490b0     0 NOTYPE  GLOBAL DEFAULT  ABS _end

No version information found in this file.
```

8 libmiASMaelf LIBRARY

As was seen in the previous section, there exists a great deal of book keeping to be followed while generating a relocatable file. **libmiasmaelf** essentially takes care of this book

keeping[9].

Why the Schism between miASMa and libmiasmaelf?

The reasons for this are

- To clearly demarcate the Lexing/Parsing routines, from actual object file creation.
- By separating the two, we have created a library which can be used in other applications as well.

It must be pointed out that **miASMa** uses this library for generating the machine code. Let us now have a look at how one can use the library, to **write a relocatable file** directly, and not having to look at **miASMa** for the time being.

```
/* hello.c */
#include <iostream>
#include <vector>
#include "libmiasmaelf.h"

int main(void )
{
  char text[] = {
    '\xB8', '\x04', '\x00', '\x00', '\x00',   // mov eax, 4
    '\xBB', '\x01', '\x00', '\x00', '\x00',   // mov ebx, 1
    '\xB9', '\x00', '\x00', '\x00', '\x00',   // mov ecx, myvariable
    '\xBA', '\x0E', '\x00', '\x00', '\x00',   // mov edx, 14
    '\xCD', '\x80',                           // int 0x80
    '\xB8', '\x01', '\x00', '\x00', '\x00',   // mov eax, 1
    '\xCD', '\x80'                            // int 0x80
  };
```

[9]Where does .text section fall, where are its contents?

```
char data[] = {
  '\x48', '\x65', '\x6C', '\x6C', '\x6F',
  '\x2C', '\x20', '\x57', '\x6F', '\x72',
  '\x6C', '\x64', '\x21', '\x0A'
}; //Hello,World!

vector<char> vtext(&text[0], &text[29]);
vector<char> vdata(&data[0], &data[14]);
miasmaELF obj;

obj.InitializeELFHeader();
obj.InitializeSymbolTable();

obj.AddNewSection(".shstrtab",SHT_STRTAB,  0,0,0,0,0,0);
obj.AddNewSection(".text",    SHT_PROGBITS,6,0,0,0,16,0);
obj.AddNewSection(".data",    SHT_PROGBITS,3,0,0,0,16,0);
obj.AddNewSection(".symtab",  SHT_SYMTAB,  0,0,
  obj.GetSectionIndexOfType(SHT_STRTAB, ".strtab"),
  0,
  4,sizeof(Elf32_Sym));

obj.AddNewSection(".rel.text",SHT_REL,0,0,
  obj.GetSectionIndexOfType(SHT_SYMTAB),
  obj.GetSectionIndexOfType(SHT_PROGBITS, ".text"),
  4,sizeof(Elf32_Rel));

obj.AddContents(vtext, obj.GetSectionIndexOfType(SHT_PROGBITS,".text"));
obj.AddContents(vdata, obj.GetSectionIndexOfType(SHT_PROGBITS,".data"));
```

```
  obj.AddSymbol("_start",0,0, STB_WEAK, STT_FUNC,
obj.GetSectionIndexOfType(SHT_PROGBITS, ".text"));
  obj.AddSymbol("myvariable",0,0, STB_GLOBAL, STT_OBJECT,
obj.GetSectionIndexOfType(SHT_PROGBITS, ".data"));

  obj.AddRelocationEntry(11, obj.ReturnSymbolIndex("myvariable"),
 R_386_RELATIVE,
 obj.GetSectionIndexOfType(SHT_REL, ".rel.text"));

  obj.PrepareFile();
  obj.WriteFile("hello.o");
}
```

The library makes extensive use of **vectors** - a data structure that is a part of the **Standard Template Library**. We first create the machine language equivalents of every instruction and populate the vectors accordingly.
We then initialize the ELFHeader, the SymbolTable Initialization follows next. This is done after defining an object of type miasmaELF.
We then go on to initialize individual sections. The function
obj.GetSectionIndexOfType(SHT_PROGBITS, ".text")
is used when one wants to obtain the SectionIndex of a given section. We find this function helps greatly in linking the various structures that are described in **elf.h**. Here, it is used in building the section header of a particular section. It is imperative that the user of the library must have a general idea of the various structures that are involved.
We then invoke
obj.AddContents(vtext, obj.GetSectionIndexOfType(SHT_PROGBITS,".text"));
which add the contents to the text section. The 2nd argument to AddContents is the section that we are referring to. In this case it is the .text section, and from our example the Index=3.

We employ a similar technique to add Symbols and Relocation Entries.
To finally write the file one must first must prepare it by invoking the function Prepare-File(...), and only then invoke WriteFile(FileName)
To compile hello.c one must link it with libmiasmaelf.o

9 IMPLEMENTATION

The source code has been attached in the appendix.

9.1 EXAMPLE PROGRAMS THAT CAN BE ASSEMBLED WITH MIASMA

Excuse us for our poor selection of variable names.

9.1.1 BUBBLE SORT

```
SEGMENT .DATA
ARR DB 'JIHGFEDCBA'
SIZE DB 10
SEGMENT .TEXT
MOV DX,0
OUTER:
CMP DX,9
JE EXITO
MOV BX,0
INNER:
MOV CX,10
SUB CX,DX
DEC CX
CMP BX,CX
JE EXITI
```

```
MOV AL,[ARR+EBX+1]
CMP [ARR+EBX+0],AL
JLE OK
MOV AL,[ARR+EBX+1]
XCHG [ARR+EBX+0],AL
MOV [ARR+EBX+1],AL
OK:
INC BX
JMP INNER
EXITI:
INC DX
JMP OUTER
EXITO:
MOV AX,4
MOV BX,1
MOV ECX,ARR
MOV DX,10
INT 80H
MOV AX,1
MOV BX,0
INT 80H
```

9.1.2 FACTORIAL

```
SEGMENT .DATA
NUM DB '1'
RES DB '1'
SEGMENT .TEXT
MOV AX,3
MOV BX,0
```

```
MOV ECX,NUM
MOV DX,1
INT 128
MOV DX,30H
MOV AL,[RES]
SUB AX,DX
MOV [RES],AL
MOV AL,[NUM]
SUB AX,DX
CALL FACT
AAM
ADD AL,30H
ADD AH,30H
MOV [NUM],AH
MOV [RES],AL
MOV AX,4
MOV BX,1
MOV ECX,NUM
MOV DX,2
INT 128
MOV AX,1
MOV BX,0
INT 80H
FACT:
CMP AX,0
JE RETURN
PUSH AX
DEC AX
CALL FACT
```

```
POP AX
MUL BYTE[RES]
MOV [RES],AL
RETURN:
RET
```

9.1.3 FIBONACCI SERIES

```
SEGMENT .DATA
NUMA DB 0
NUMB DB 1
NUMC DB 2
NUMD DB 0
SEGMENT .TEXT
MOV SI,9H
HI:
MOV AX,[NUMB]
MOV CX,[NUMA]
MOV [NUMA],AX
ADD AX,CX
MOV [NUMB],AX
AAM
ADD AX,3030H
MOV [NUMC],AH
MOV [NUMD],AL
MOV AX,4
MOV BX,1
MOV ECX,NUMC
MOV DX,2
INT 80H
```

```
DEC SI
CMP SI,0
JNE HI
MOV AX,1
MOV BX,0
INT 80H
```

9.1.4 nCr COMBINATION

```
SEGMENT .DATA
N DB 06H
R DB 04H
RES DB 00H
RESA DB 00H
SEGMENT .TEXT
MOV AL,[N]
MOV BL,[R]
CALL NCR
MOV AL,[RES]
AAM
ADD AX,3030H
MOV [RES],AH
MOV [RESA],AL
MOV AX,4
MOV BX,1
MOV ECX,RES
MOV DX,2
INT 80H
MOV AX,1
MOV BX,0
```

```
INT 80H
NCR:
CMP AL,BL
JE INCA
CMP BL,0
JE INCA
CMP BL,1
JE ADDN
PUSH AX
DEC AL
CMP AL,BL
POP AX
JE ADDN
DEC AL
PUSH AX
PUSH BX
CALL NCR
POP BX
POP AX
DEC BL
PUSH AX
PUSH BX
CALL NCR
POP BX
POP AX
RET
INCA:
INC BYTE[RES]
RET
```

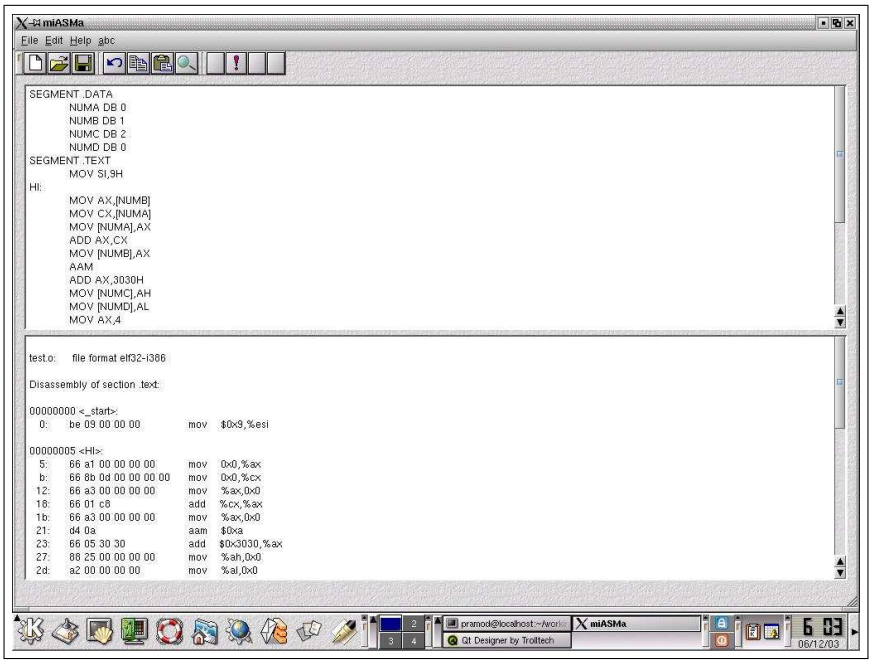

Figure 11: objdump as seen in the Integrated Development Environment

```
ADDN:
ADD [RES],AL
RET
```

9.2 SCREEN SHOTS

Here we have attached a few screen shots of the Integrated Development Environment (IDE) that we have created for **miASMa**.

10 SYSTEM TESTING

In the creation of **miASMa**, a top-down and incremental approach to testing has been followed. Initially, a functional product that could assemble a small set of instructions was developed. This was thoroughly tested for errors. Once this was done, a framework for the extension of the instruction set was established. The instructions to be added were

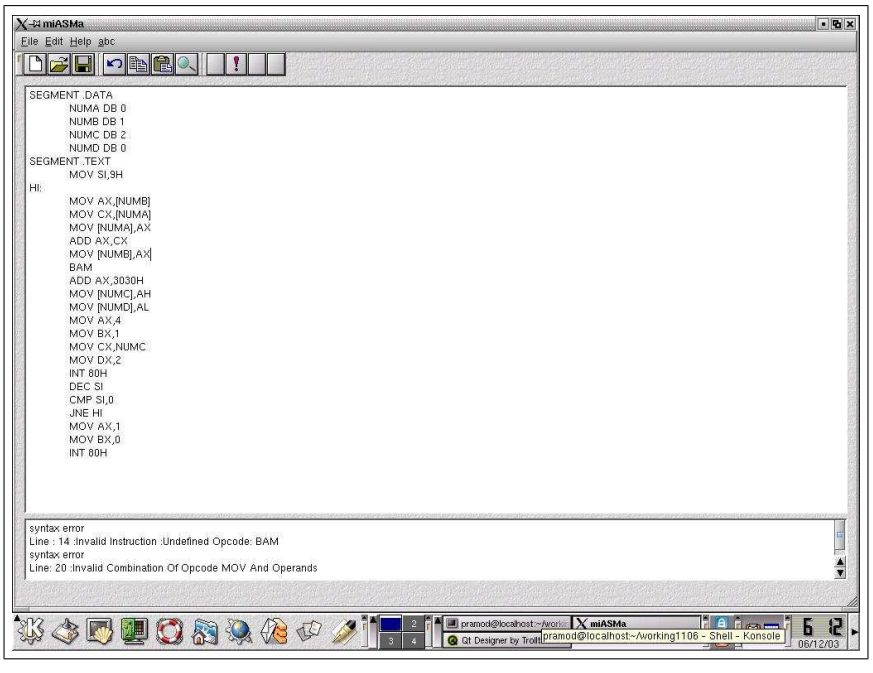

Figure 12: Assembly errors as seen in the IDE

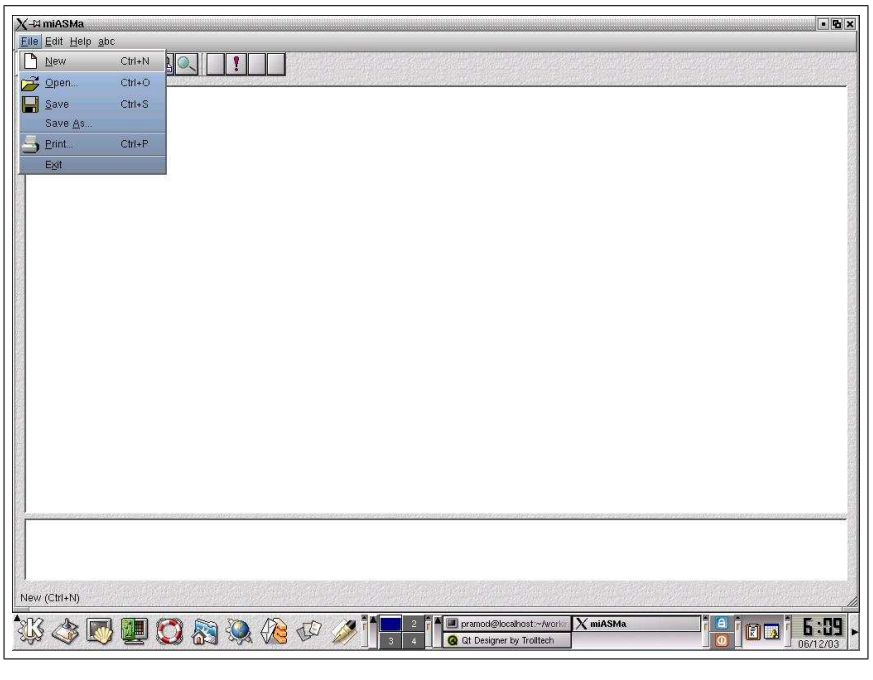

Figure 13: The MenuBar as seen in the IDE

later tested individually and then integrated with the system.

10.1 UNIT TEST REPORT

10.1.1 DRIVERS AND STUBS

The use of a top-down approach necessitated the building of stubs. Stubs are dummy modules used to simulate the working of certain modules that are required but have not yet been developed during the testing of a higher level module. *For instance, in order to test the creation of an object file in the correct ELF format that could be input to LD (linker), the parsing routine was replaced by a dummy routine that added a predefined set of object records to create the object file.*

10.2 TEST SUMMARY

10.2.1 MODULE-0

NAME:miASMa
OTHER MODULES IN THIS TESTING:macro preprocessor, pass_1 and pass_2.
DESCRIPTION: To assemble and execute the source program.
ERRORS DETECTED AND CORRECTION: This is the top most level module. The errors that were encountered during the testing of this module were basically those errors that were introduced during the integration of new modules. Since such modules were tested prior to the integration, the errors introduced were minimal.

10.2.2 MODULE-1

NAME:macro preprocessor
DESCRIPTION: To expand the user defined macro into a separate file.
ERRORS DETECTED AND CORRECTION: During the testing of the macro preprocessor, errors were detected in the identification of arguments passed to the macro.These were corrected by restructuring the scanner (lexer) for the macro processor.

10.2.3 MODULE-2

NAME:pass_1 and pass_2
OTHER MODULES IN THIS TESTING:A lexer called scan.l which handles the recognition and classification of the tokens and parsers pass1_parse and pass2_parse which handle the recognition of the scanned tokens as some language constructs defined and accordingly update the symbol file and the object file.
DESCRIPTION: To translate mnemonic operation codes to their machine language equivalents and to replace symbols by their values.
ERRORS DETECTED AND CORRECTION: Certain errors related to the grammar were detected during the testing of this module. The grammar was reconsidered and slight modifications were made to correct these errors.

10.3 SYSTEM TEST REPORTS

An incremental testing approach was used for system integration and testing. As a first step, a basic framework for the assembler was created. This had a command-line interface and did not support macros. In the next step, support was provided to assemble more instructions. In the third step, the pretested macro processor was added. Finally, the command-line interface was changed into a mouse-based interface by integrating the assembler with a tested editor.
With this, **miASMa** now has an integrated development environment with which the programmer can write assembly language programs (with macros), edit, assemble them all at one place.

10.4 ERROR REPORTS

1 The assembler was found to create inappropriate object files when provided with a program with a few blank lines. In order to correct this, the grammar was restructured to ignore blank lines in between source lines.

2 Some errors that created illegal relocation records were detected and corrected,

which was rectified with the parallel development of the ELF library y **libmiASMaelf**

11 RESULTS AND CONCLUSION

The main ideas that we have learned from this project are:

1 On how to code by strictly conforming to a specification.

2 On how to integrate software components written by each of us, each having our own different styles, into a single cohesive unit.

3 On how to write a complete piece of sufficiently complex system software.

4 On the various Object Formats such as ELF, AOUT, COFF etc.

5 On writing assembly programs in Unix/Linux environment, when in the past, we have written all our previous assembly language programs (for our Microprocessor Laboratory) in DOS using MASM.

6 On writing the same programs (most of them!) in **miASMa**.As of today we have written programs for Fibonacci series, bubble sort etc. This was done to show that it is possible to write assembly language programs [10] in **miASMa**.

miASMa and **libmiASMaELF** will soon be hosted on freshmeat.net and maintained regularly.

References

[1] [1995] Tools Interface Standard - Executable and Linking format Specification Version 1.2

[2] [1997] Leland L Beck - System Software 3rd Edition

[10] that do not use Dynamic Linking

[3] [1997] Pankaj Jalote - An integrated approach to software engineering 2nd Edition

[4] [1994] http://www.gnu.org/manual/ld-2.9.1/html_chapter/ld_1.html - For GNU/Linux specific information

[5] [2003] http://www.trolltech.com/products/qt/ - For Qt Specific information

[6] [1999] http://www.luv.asn.au/overheads/lex_yacc/ - For a wonderful introduction to Lex and YACC

[7] [2003] NASM info and manual pages